

Morpher Wallet

This is the guideline for a security review of the Morpher Wallet

General thoughts for the wallet

In contrast to other wallets, this wallet should simply replace the *secure keystore* from wallets. It should not connect to a blockchain node, or represent a full API.

The wallet is running on its own domain with very clear inputs and outputs, so that the surface for XSS-like attacks is minimized. Running on a separate domain also ensures that there is no leakage from the `localStorage` or `sessionStore`.

Getting started quickly

The wallet comes packed with a sample docker-compose file. It also comes pre-packaged with development keys for Social Recovery. All you need to fill in is AWS keys for Email2FA functionality.

```
cp .env.aws.backend.example .env.aws.backend
vi .env.aws.backend
```

Fill in the values then simply run `docker-compose up`. It will start a postgres database, an express server backend and a vue development environment. The vue environment would be the content of the iframe.

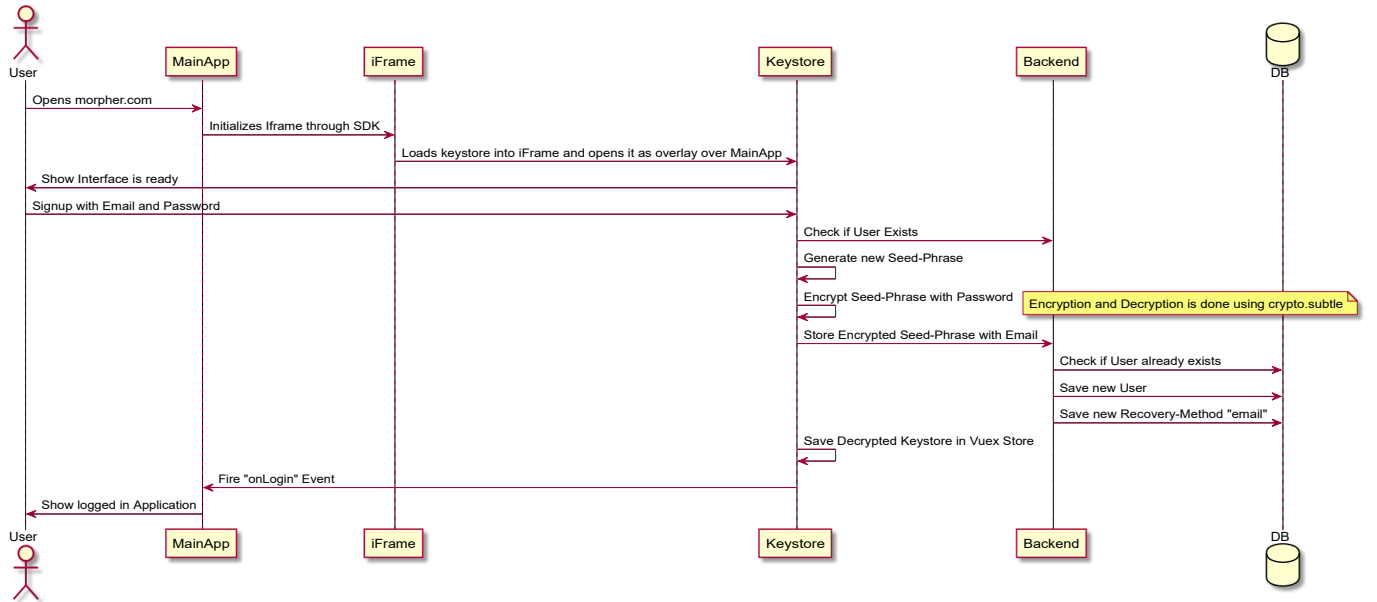
If you want to simulate a full iframe environment, please find a sample "parent-frame" application in `client_trade_morpher_com`. `npm install && npm run start`.

Iframe Flow

The Iframe and parent app follows a flow with defined interfaces for inputs and outputs.

The inputs from the main app are:

1. Get the Accounts (no payload attached)
2. Sign a Transaction (payload is the transaction object)
3. Sign a Message (payload is the message object)

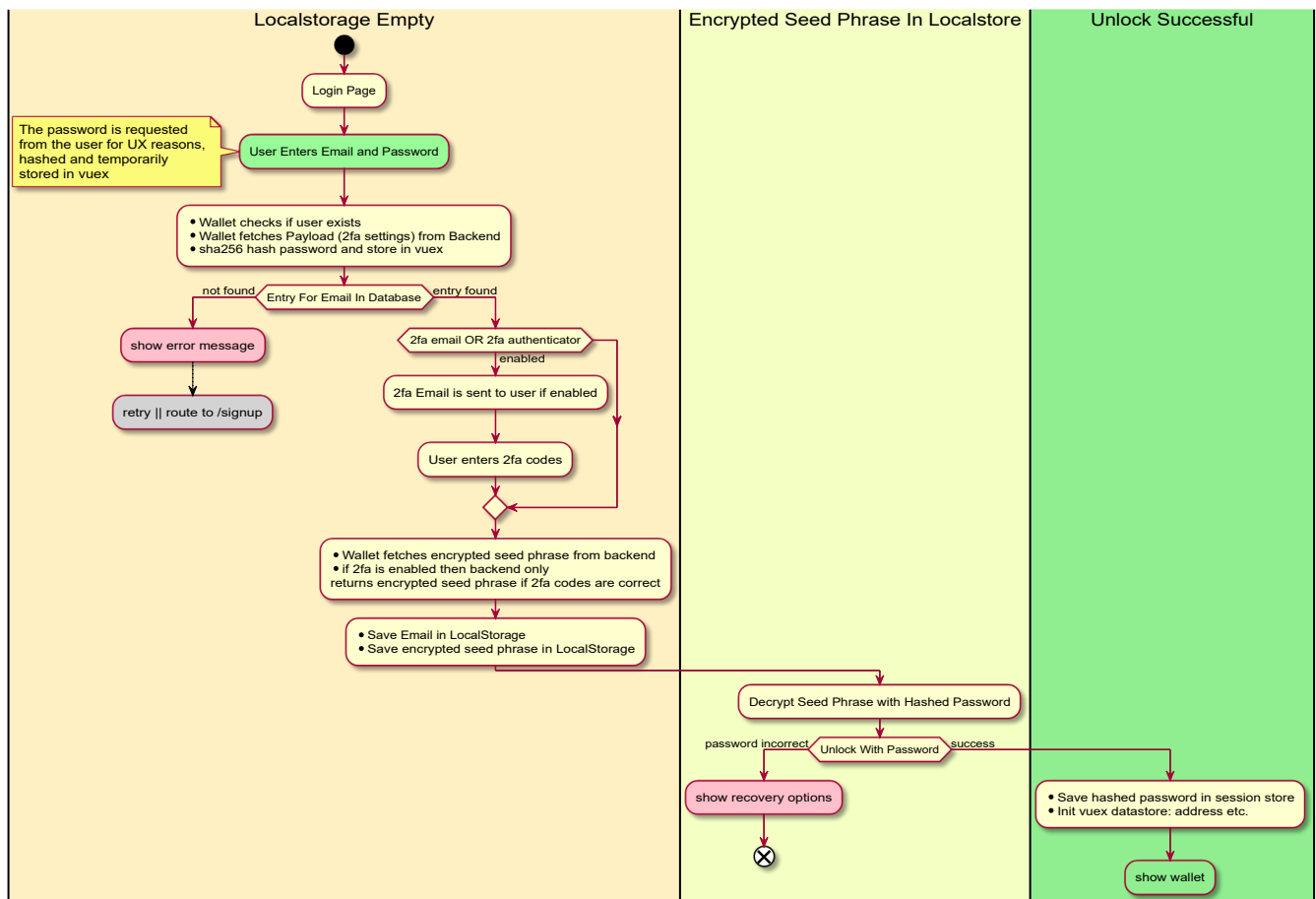


Login Flow for the Wallet

This section describes the login and unlock-Flow for the wallet.

The login and unlock flow are different only by the contents of the localStorage. If the encryptedSeed is already stored in localStorage, then we don't need to fetch it from the backend.

Login Flow



When the user logs into the wallet the first time, then the browsers localStorage and sessionStorage is empty.

1. The user is required to enter Email-Address and Password.

The password is temporarily stored, because of UX reasons. It is used later in the process.

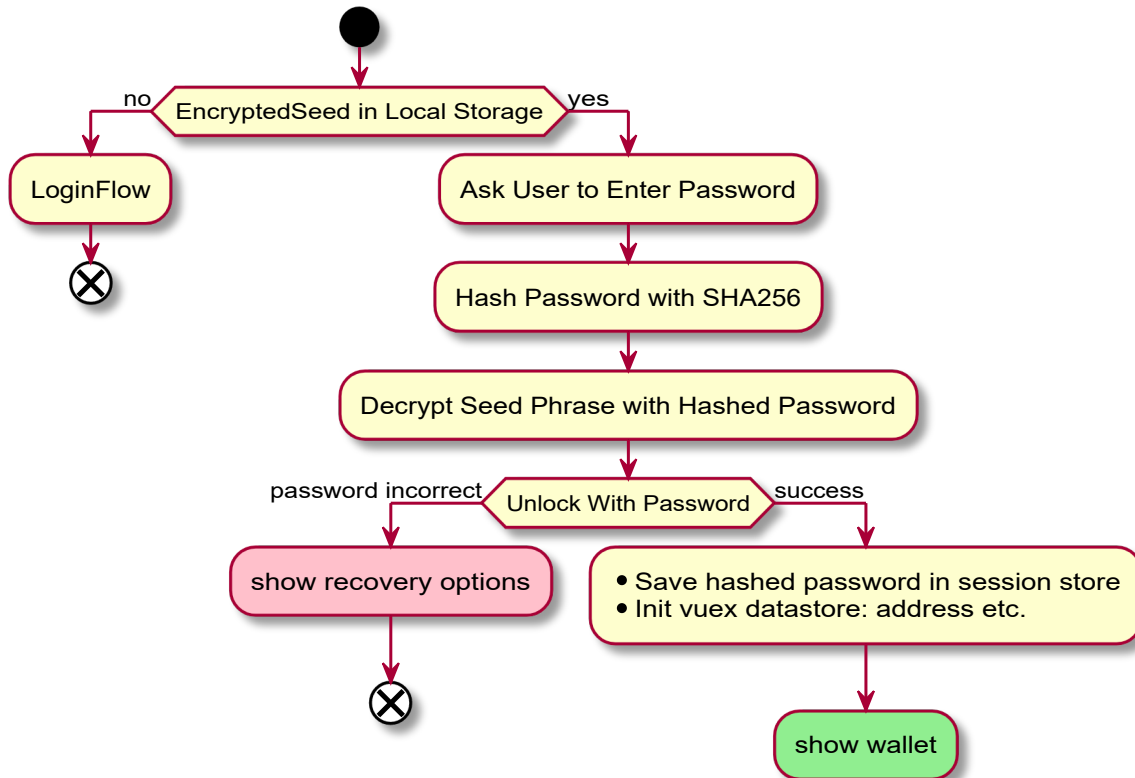
2. The Wallet checks with the backend if the user already exists.

Sidenote: It became "good behavior" to show generic error messages to avoid leaking user-exists information. This simply doesn't make sense, because the backend is a data-store for an encrypted wallet. The actual decryption process is done in-browser.

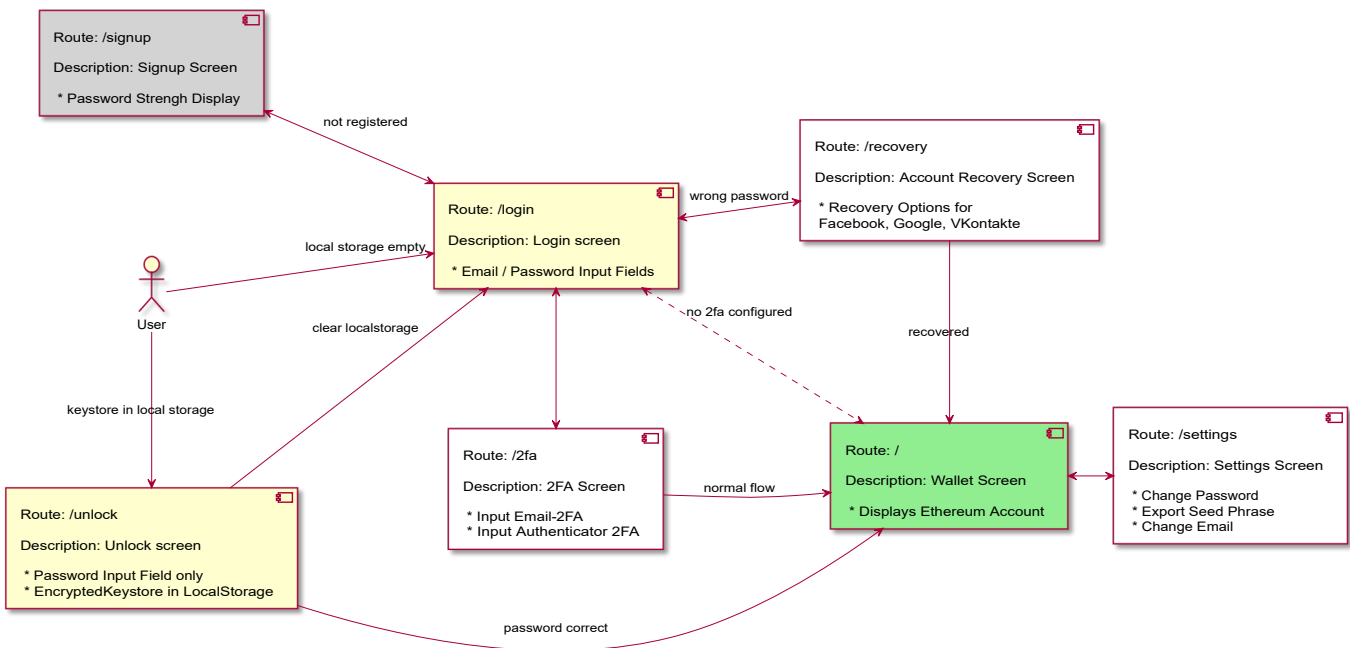
3. Depending on the 2FA settings, the user is then required to enter the 2FA codes. The backend only delivers the wallet when the 2fa codes are correct and onyl the correct password can then decrypt the keystore.

4. If decryption with the password was successful then the *password-encrypted* keystore (after decrypt with 2fa) is stored in localStorage. This lets the user "unlock" the keystore with the password only in sub-sequent uses of the wallet without re-fetching it from the server.
5. The wallet is initialized: The keystore is used to generate an account. The account is stored in the JavaScript-Storage-Backend. The frontend shows the Address and UI

Unlock Flow



Description of the Routes



Route	description
Login: /login	The first screen if the LocalStorage is empty
Signup: /signup	If the user wants to create a new Account, then this is the component which takes care of that.
Unlock: /unlock	Let's the user unlock the keystore. This is the default screen if the encrypted keystore is in local-storage. Otherwise the App will default to /login
2FA: /twoFa	If the user is required to enter 2fa codes, then he is redirected here before the keystore is fetched from the server
Wallet: /	If the user is logged in correctly, he will see the wallet address and can potentially switch accounts, sign transactions etc
Settings: /settings	Change Email, Password, Export Seed Phrases, Add Account Recovery
(Later) Sign Transaction: /signtx	Shows a summary of the transaction and a sign/cancel button

Overview

This document describes the external packages used in the Morpher Wallet.

Frontend Technologies

The frontend is split into two parts:

1. The Keystore
2. The iFrame SDK

In general both are written in TypeScript (or ported to TS).

The Keystore is written in VueJs with a Vuex datastore.

The iFrame SDK is written using a native iFrame HTML element and binding parent/child listeners through `penpal` to the parent/child document. More on the flow below.

List of Tools/Frameworks Used

The following is a non-exhaustive list of important packages used in the Keystore-App with their description:

Web3js Wallet

<https://web3js.readthedocs.io/en/v1.2.0/web3-eth-accounts.html>

The `web3.eth.accounts` contains functions to generate Ethereum accounts and sign transactions and data.

It's currently marked as *"This package has NOT been audited and might potentially be unsafe. Take precautions to clear memory properly, store the private keys safely, and test transaction receiving and sending functionality properly before using in production!"*. We're aware of this fact, no data leakage is indicated at this point.

Web3js is a well maintained open source library and widely in use. `Web3.eth.accounts` is giving back an object containing scoped functions to sign a transaction. The object is never exposed

```
{
  address: "0xb8CE9ab6943e0eCED004cDe8e3bBed6568B2Fa01",
  privateKey:
    "0x348ce564d427a3311b6536bbcff9390d69395b06ed6c486954e971d960fe8709",
  signTransaction: function(tx){...},
  sign: function(data){...},
  encrypt: function(password){...}
}
```

Web3js can either import private keys or generate them. Since we base the wallet on *seed-phrases*, another library, `bip39`, is used to generate the seed phrases.

bip39

<https://www.npmjs.com/package/bip39>

JavaScript implementation of Bitcoin BIP39: Mnemonic code for generating deterministic keys

```
const mnemonic = bip39.generateMnemonic()  
// => 'seed sock milk update focus rotate barely fade car face mechanic mercy'
```

This generates a mnemonic using [crypto.getRandomValues](#) in the browser and then fetches words from an english wordlist. The [wordlist](#) is 2048 words long and the entropy generated by default is 128 bits. This gives 16 random bytes using the [randombytes](#) package.

This mnemonic is then piped through ethereumjs-utils to generate a private key.

hdkey from ethereumjs-utils

<https://www.npmjs.com/package/ethereumjs-util> is a collection of utility functions for Ethereum.

The Morpher Wallet uses the [hdkey](#) functions from ethereumjs-wallet

We use it in `vue/src/utils/keystore.ts` to generate the private keys. Sample code:

```
import { hdkey } from 'ethereumjs-wallet';  
function getPrivateKeyFromMnemonic(mnemonic: string, index: number) {  
  
  const seed = mnemonicToSeedSync(mnemonic);  
  const hdwallet = hdkey.fromMasterSeed(seed);  
  const walletHdPath = "m/44'/60'/0'/0/";  
  
  const wallet = hdwallet.derivePath(walletHdPath + index).getWallet();  
  
  const privateKey = wallet.getPrivateKey().toString("hex");  
  
  return privateKey;  
}
```

Wallet Settings

This describes the flow for the settings.

Middleware

All writing operations for user-settings are bound to a middleware. All requests are signed using the private key of the user as secret and the complete request-payload including a one-time nonce as message content.

Request Signing

To authenticate a stateless request to the backend to update user information, the requests are signed.

The signature method is very similar to HMAC signatures used by Amazon.

In general it prevents:

1. MITM Attacks by signing the payload
2. Replay Attacks by introducing a nonce

Nonce

The nonce is a known number that increases every time an authenticated request is sent to the backend. No nonce is ever the same. The nonce is part of the payload.

Payload

When the payload for POST or GET requests is sent, then an additional header is added to the request with the signature.

Signature

The signature is generated by using the private key of the user. The eth-address for the user is stored backend-side in the database. In general, every authenticated request must fulfill the following equation:

```
eth_address_stored_in_backend == ec_recover(request.headers.signature,  
request.body.payload);
```

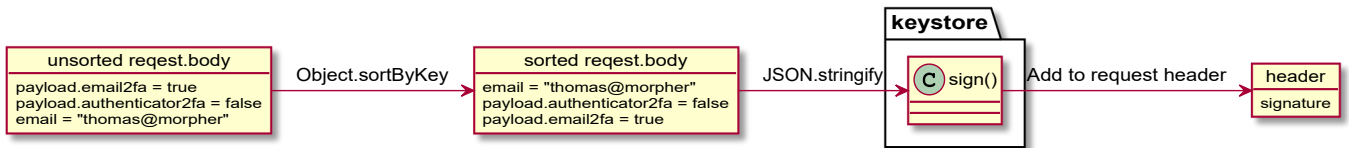
Otherwise it will fail.

To recover the eth_address we use [ethereumjs-util.ecrecover](#).

Client Side Signature Creation

The signature is created client-side with the keystore (`web3.eth.accounts.wallet.sign`).

In general it follows this flow:



The following pseudo-code is responsible for the signature creation:

```

function addSignature(requestObject) {
  requestObject.body.nonce = getNonceFromBackend();

  /**
  requestObject.body e.g.:
  {
    email: "thomas@morpher.some",
    payload: {
      email2fa: true,
      authenticator2fa: false
    },
    nonce: 12345
  }
  */
  data_sorted = sortAlphabeticallyByKey(requestObject.body)
  //watch out that also sub-objects are sorted! So it looks like this:
  data_sorted = {
    email: ...,
    nonce: ...,
    payload: {
      authenticator2fa: ...,
      email2fa: ...
    }
  }

  //then stringify it - either like this:
  signedMessage = keystore.sign(JSON.stringify(data_sorted))

  requestObject.headers.signedMessage = signedMessage;
  requestObject.headers.signatureKey = sha256(email);
  return requestObject
}

//here the function calls the backend and adds the signature
fetch('/backendUrl', addSignature(requestObject));
  
```

Backend

On the backend a middleware is checking the request body against the users eth_address after ec_recovery of the signature:

In Pseudocode:

```

recovery = findOne({
  where: {
    "key": req.headers.signatureKey
  }, include: User)
if (recovery) {
  eth_address = ecrecover(JSON.stringify(sortAlphabetically(req.body)),
req.headers.signedMessage)
  if (eth_address == recovery.user.eth_address) {
    //SUCCESS HERE
    //routing continues
  }
}

throw 505-error //happens in any other case.

```

Password Change

To change the password, the user has to provide the old password and repeat the new password twice.

The same validation logic as in signup applies to the password change mechanism. It must fulfill:

1. Minimum length
2. Type of characters
3. Combination of upper/lower case

If the validation passes then the users

1. keystore is decrypted using the old-password.
2. re-encrypted using the new password
3. the encrypted keystore is sent to the server to replace the old keystore
4. server-side the keystore is encrypted with the server-side keys

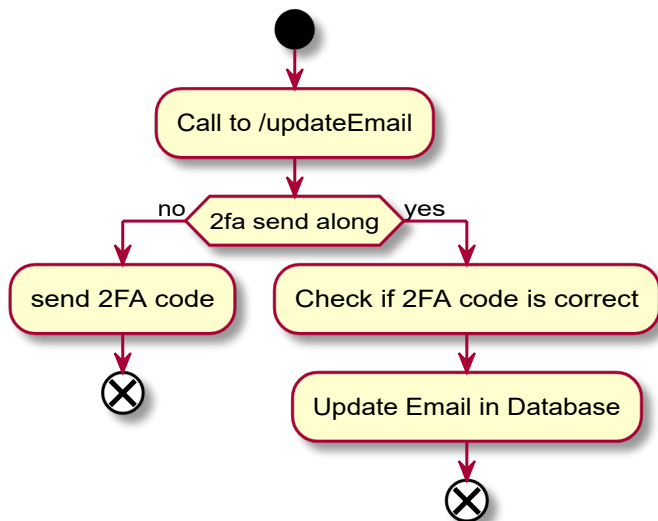
Email Change

Changing the Email Address is a more complex topic than it appears on the surface.

Pre-assumptions:

- The user has successfully unlocked his keystore and has an unlocked wallet in localStorage.
- The user might enter a wrong *new* email address, which needs to be validated.

Possible attack vector: Someone having access to the users computer might maliciously try to change the users email address.



On the first request a 2FA code is generated and sent to the *new* email address.

The user is then requested to enter the 2FA code - to validate that he really is the owner of the new email address - and send it along with the payload.

Note: It is clear that writing a different, un-validated, new email-address potentially *can* be circumvented on a lower level by grabbing a 2fa code from the first request and change the target email address on the subsequent request. The additional layers are for ux reasons and sanity-checks, rather than bullet-proof security. As soon as the user is in possession of an unencrypted keystore it is to be expected that changing an email address is the least of the problems and the intention of the user.

Logging for the Wallet

There are two separate logging mechanisms built into the wallet.

1. Audit logs, which have unlimited retention
2. User Interaction logs, which have limited retention.

Audit Logs

We are aware that operating a wallet on behalf of users, even without having access to the underlying keystore, is bound to a lot of responsibility. A keystore can eventually control very large sums of funds. If there are user-errors or malfunctions during updating, e.g. user email addresses etc, a user can easily lose his keys forever. To circumvent this, we build an audit log, so that we can track users actions and eventually help users in case of malfunctions, user errors or other problems. This is done via a database-table called "Userhistory" which logs only the most profound changes on user data, for example changing the mail address.

User Interaction Log

To understand users better, we log almost any action to cloudwatch. This can be fetching a wallet, validating email/password or validating 2fa codes. The cloudwatch logs have a limited retention, no sensitive information like unencrypted keys or passwords, and should only help understand any user-errors or malfunctions better.